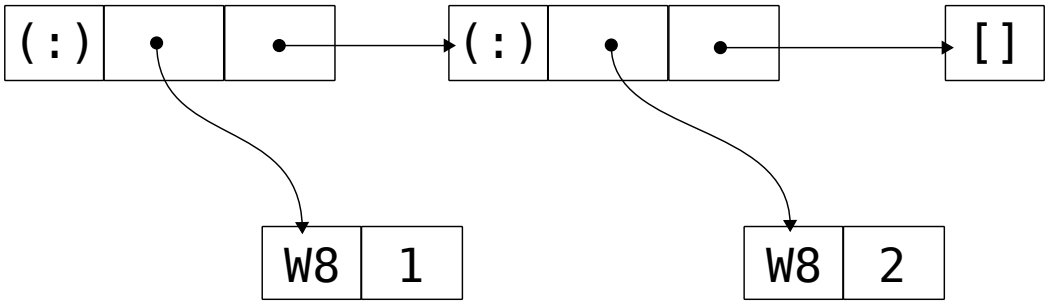
Hatékony Input/Output

A listák nem eléggé hatékonyak...

A szabványos listák nem ideálisak input/outputra:

- Sok a mutató, az ún. *cache miss*
- Sok helyet foglalnak, nagy az ún. *overhead*



Miként is fog ez pontosan működni Haskellben?

$$
\begin{aligned}
&\mathit{example} = \mathbf{do} \\
&\quad \mathit{contents} \leftarrow \mathit{readFile}\ \text{"}/\mathit{etc}/\mathit{hosts}\text{"} \\
&\quad \mathit{return}\ \$\ \mathit{head}\ \$\ \mathit{lines}\ \mathit{contents}
\end{aligned}
$$

A *readFile* működése

```
readFile :: FilePath → IO String
readFile name = openFile name ReadMode >>= hGetContents

hGetContents :: Handle → IO String
hGetContents handle =
  wantReadableHandle "hGetContents" handle $ λ h . do
    xs ← lazyRead handle
    return (h { haType = SemiClosedHandle }, xs)
-- unsafeInterleaveIO: az IO-akció csak az érték hivatkozásakor
-- (késleltetten) hajtódik végre
lazyRead :: Handle → IO String
lazyRead handle = unsafeInterleaveIO $
  withHandle "hGetContents" handle $ λ h . do
    case (haType h) of
      SemiClosedHandle → lazyReadBuffered handle h
      ClosedHandle     → ioException $
        IOError (Just handle) IllegalOperation "hGetContents"
          "delayed read on closed handle" Nothing Nothing
      _ → ioException $
        IOError (Just handle) IllegalOperation "hGetContents"
          "illegal handle type" Nothing Nothing
```

Data.ByteString

`http://hackage.haskell.org/package/bytestring`

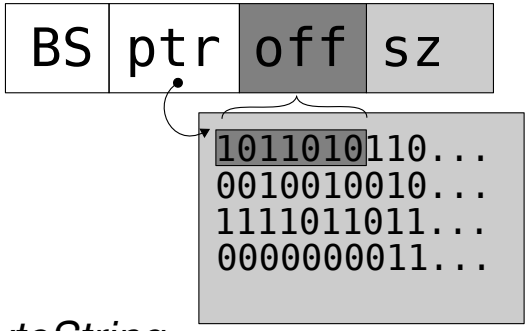
Idő- és tárhatékony adatszerkezet, byte-ok véges (*finite sequence*) és végtelen sorozatához (*infinite stream*).

- *Mohó (strict) ByteString.* Egyetlen összefüggő memóriaterület, tömb. C és Haskell közti kommunikáció esetén kényelmes.
- *Lusta (lazy) ByteString.* Nagyobb összefüggő memóriaterületek lustán láncolt listája. Input/ouput adatfolyamok esetén hasznos.

```
import qualified ByteString as B

example = do
  contents ← B.readFile "/etc/hosts"
  return $ B.head $ B.lines contents
```

ByteString: Mohó változat (definíció)

```
{-# LANGUAGE BangPatterns #-}
import Foreign

-- minden ByteString költsége: 9 gépi szó
data ByteString = BS
  {-# UNPACK #-} !(ForeignPtr Word8) -- tartalom (puffer)
  {-# UNPACK #-} !Int                -- kezdőpozíció (offset)
  {-# UNPACK #-} !Int                -- hossz

empty :: ByteString
empty = BS nullForeignPtr 0 0

singleton :: Word8 → ByteString
singleton c = unsafeCreate 1 $ λ b . poke b c

unsafeCreate :: Int → (Ptr Word8 → IO ()) → ByteString
unsafeCreate l f = unsafePerformIO $ create l f

create :: Int → (Ptr Word8 → IO ()) → IO ByteString
create l f = do
  fp ← mallocForeignPtrBytes l
  withForeignPtr fp $ λ p . f p
  return $! BS fp 0 l
```



ByteString: Mohó változat (definíció)

```
pack :: [Word8] → ByteString
pack ws = unsafeCreate (length ws) $ λ p . f p ws
  where
    f !_ []          = return ()
    f !p (x : xs)    = poke p x >> f (p `plusPtr` 1) xs

unpack :: ByteString → [Word8]
unpack bs = unpackAppend bs []

unpackAppend :: ByteString → [Word8] → [Word8]
unpackAppend (BS fp off len) xs
  | len ≤ 100  = unpackAppendS (BS fp off len) xs
  | otherwise  = unpackAppendS (BS fp off 100) remainder
  where
    remainder = unpackAppend (BS fp (off + 100) (len − 100)) xs

unpackAppendS :: ByteString → [Word8] → [Word8]
unpackAppendS (BS fp off len) xs = unsafePerformIO $ withForeignPtr fp $
  λ base . loop (base `plusPtr` (off − 1))
                (base `plusPtr` (off − 1 + len)) xs
  where
    loop !sentinal !p acc
      | p ≡ sentinal  = return acc
      | otherwise     = do
        x ← peek p
        loop sentinal (p `plusPtr` (−1)) (x : acc)
```

ByteString: Mohó változat (műveletek)

```
{-# LANGUAGE OverloadedStrings #-}
class IsString α where
  fromString :: String → α

class Show α where
  show :: α → String

instance IsString ByteString where fromString = packChars
instance Show ByteString where show = unpackChars

take :: Int → ByteString → ByteString
take n bs@(BS ptr off len)
  | n ≤ 0      = empty
  | n ≥ len    = bs
  | otherwise  = BS ptr off n

append :: ByteString → ByteString → ByteString
append (BS _ _ 0)          y                    = y
append x                   (BS _ _ 0)           = x
append (BS ptr1 off1 len1) (BS ptr2 off2 len2) =
  unsafeCreate (len1 + len2) $ λ destptr1 . do
    let destptr2 = destptr1 `plusPtr` len1
    withForeignPtr ptr1 $ λ p2 . memcpy destptr1 (p1 `plusPtr` off1) len1
    withForeignPtr ptr2 $ λ p2 . memcpy destptr2 (p2 `plusPtr` off2) len2
```

ByteString: Lusta változat

Mohó (egyenként legfeljebb 64 KB méretű) *ByteString* értékek lustán láncolt listája.

$$
\begin{aligned}
&\textbf{data}\ \mathit{LazyByteString}\\
&\quad = \mathit{Empty}\\
&\quad \mid\ \ \mathit{Chunk}\ !\mathit{Bytestring}\ \mathit{LazyByteString}
\end{aligned}
$$

- Byte-ok tiszta folyamának hatékony és tömör ábrázolási módja. A nem hivatkozott szeleteket a szemétgyűjtögető felszabadítja.
- Az adatfolyamok feldolgozása konstans memóriahasználattal jár, de csak egyszerű, rövidéletű programok esetében javasolt.
- A szeletek határán kisebb a hatékonyság.
- A hibákat nem lehet egyértelműen jelezni.

ByteString: Lusta változat (műveletek)

```
import qualified ByteString as BS

empty :: LazyByteString
empty = Empty

singleton :: Word8 → LazyByteString
singleton w = Chunk (BS.singleton w) Empty

chunk :: ByteString → LazyByteString → LazyByteString
chunk c@(BS.BS _ _ len) cs
  | len ≡ 0   = cs
  | otherwise = Chunk c cs

pack :: [Word8] → LazyByteString
pack ws = f 32 ws
  where
    f n cs = case (BS.packUptoLenBytes n cs) of
      (bs, [])    → chunk bs Empty
      (bs, cs′)   → Chunk bs $ f ((n ∗ 2) `min` optimalChunkSize) cs′

    optimalChunkSize = 4 ∗ 1024

unpack :: LazyByteString → [Word8]
unpack Empty          = []
unpack (Chunk c cs)   = BS.unpackAppend c (unpack cs)
```

Data.ByteString.Builder

Az ismételt *append* nagyon költséges. Ezért a lusta *ByteString* értékekből készítsünk egy differencialistát:

- konstans idejű *append* műveletet kapunk
- rögzített vagy akár változó byte-hosszúságú értékeket tudunk *ByteString* típusúra alakítani
- alkalmazástól függő módon állíthatunk elő *ByteString* értéket

```
import qualified Data.ByteString.Lazy as L
import Data.ByteString.Builder
import Data.Foldable (foldMap)
import Data.Monoid

renderString :: String → Builder
renderString cs = charUtf8 '"' ⊕ foldMap escape cs ⊕ charUtf8 '"'
  where
    escape '\\' = charUtf8 '\\' ⊕ charUtf8 '\\'
    escape '\"' = charUtf8 '\\' ⊕ charUtf8 '\"'
    escape c    = charUtf8 c

encodeUtf8 :: String → L.ByteString
encodeUtf8 = toLazyByteString ∘ renderString
```

Data.ByteString.Builder: Implementáció (részlet)

```
newtype Builder = Builder (∀ ρ . BuildStep ρ → BuildStep ρ)

-- BufferRange: Mutatók által megadott terület a memóriában.
type BuildStep α = BufferRange → IO (BuildSignal α)

data BuildSignal α
  = Done
      {-# UNPACK #-} !(Ptr Word8) -- mutató az adat után
                     α            -- állapot
  | BufferFull                    -- megtelt puffer jelzése
      {-# UNPACK #-} !Int         -- minimum szükséges méret
      {-# UNPACK #-} !(Ptr Word8) -- mutató az adat után
                     (BuildStep α) -- következő lépés
  | InsertChunk                   -- darab beillesztése
      {-# UNPACK #-} !(Ptr Word8) -- mutató az adat után
      {-# UNPACK #-} ByteString   -- a beillesztendő darab
                     (BuildStep α) -- következő lépés

append :: Builder → Builder → Builder
append (Builder b1) (Builder b2) = Builder $ b1 ∘ b2

toLazyByteString :: Builder → LazyByteString
toLazyByteString = toLazyByteStringWith
  (safeStrategy smallChunkSize {- 4 KB -} defaultChunkSize {- 32 KB -}) Empty
```

Egy hatékonyabb megoldás: *Iteratee*

Az "iteratee" egy *kompozicionális* absztrakció, amellyel szekvenciálisan gyártott bemenetet tudunk lépésről lépésre feldolgozni.

Háromféle értékét kaphat:

- A bemenet következő szelete
- „Nincs több adat”
- „Iterálás vége”

Háromféle értéket (állapotot) adhat vissza (a hívó felé):

- „Megállás egy végeredménnyel”
- „Feldolgozás folytatása”
- „Hiba történt”

Iteratee: Definíció

```
import Control.Exception
import qualified Data.ByteString as BS
import qualified Data.ByteString.Lazy.Char8 as Lazy

data Chunk = Chunk
  { chunkData    :: !Lazy.ByteString
  , chunkAtEOF   :: !Bool
  }
newtype Iteratee α = Iteratee (Chunk → Result α)

data Result α
  = Done { rResult :: α, rResidual :: Chunk }
  | NeedInput !(Iteratee α)
  | NeedIO !(IO (Result α))
  | Failed !SomeException
```

Iteratee: Sorok beolvasása (példa)

```
consumeLine :: Iteratee (Maybe Lazy.ByteString)
consumeLine = Iteratee (f Lazy.empty)

f :: Lazy.ByteString → Chunk → Result (Maybe Lazy.ByteString)
f acc (Chunk { chunkData = input, chunkAtEOF = eof })
  | ¬ (Lazy.null b) =
    Done { rResult    = Just acca
         , rResidual  = Chunk { chunkData    = Lazy.tail b
                              , chunkAtEOF   = eof
                              }
         }
  | ¬ eof = NeedInput (Iteratee (f acca))
  | otherwise =
    Done { rResult    = Nothing
         , rResidual  = Chunk { chunkData    = acca
                              , chunkAtEOF   = eof
                              }
         }
  where
    (a, b) = Lazy.break (≡ '\n') input
    acca   = acc `Lazy.append` a
```

Enumerator: Állomány feldolgozása (példa)

```
type Enumerator α = Iteratee α → IO (Result α)

enumerateFile :: FilePath → Enumerator α
enumerateFile path = λ iter .
  bracket (openFile path ReadMode) hClose (g iter)

g :: Iteratee α → Handle → IO (Result α)
g (Iteratee iter) h = do
  input ← BS.hGetSome h 32752
  if (BS.null input)
    then return $ NeedInput (Iteratee iter)
    else run $
      iter (Chunk { chunkData    = Lazy.fromChunks [input]
                  , chunkAtEOF   = False
                  })
  where
    run (NeedInput i) = g i h
    run (NeedIO i)    = i >>= run
    run result        = return result
```

Iteratee: Minden együtt

```
chunkEOF :: Chunk
chunkEOF = Chunk { chunkData = Lazy.empty, chunkAtEOF = True }

getResult :: Result α → IO α
getResult (Done { rResult = x }) = return x
getResult (NeedInput (Iteratee f)) = getResult (f chunkEOF)
getResult (NeedIO io)              = io >>= getResult
getResult (Failed e)               = throwIO e
```

```
example = do
  result ← enumerateFile "/etc/hosts" consumeLine
  getResult result
```

Text: *ByteString* szövegekre

`http://hackage.haskell.org/package/text`

Unicode karaktersorozatok és azok kódolásainak hatékony, tömör, *immutable* (mohó és lusta) ábrázolása, komoly *loop fusion* támogatással.

```
import Data.Text as T
import qualified Data.Text.IO as T
import Data.Text.Encoding as E
import Data.ByteString (ByteString)

-- „subject to fusion”: köztes adatszerkezetek elhagyása (hatékony)
countChars :: ByteString → Int
countChars = T.length ∘ T.toUpper ∘ E.decodeUtf8

example = do
  contents ← T.readFile ”/etc/hosts”
  return $ T.head $ T.lines contents
```

Text: Implementáció (bepillantás)

```
length :: Text → Int
length = streamLength ∘ stream

data Step σ α = Done | Skip !σ | Yield !α !σ
data Stream α = ∀ σ . Stream (σ → Step σ α) -- léptető függvény
                             !σ             -- aktuális állapot

stream :: Text → Stream Char
stream (Text arr off len) =
  Stream next off (betweenSize (len `shiftR` 1) len)
  where
    !end = off + len
    next !i
      | i ≥ end                  = Done
      | 0xD800 ≤ n1 ≤ 0xDBFF     = Yield (U16.chr2 n1 n2) (i + 2)
      | otherwise                = Yield (unsafeChr n1) (i + 1)
    where
      [n1, n2] = map (A.unsafeIndex arr) [i, i + 1]

streamLength :: Integral α ⇒ Stream Char → α
streamLength (Stream next s0 _len) = loop_length 0 s0
  where
    loop_length !z s = case (next s) of
      Done      → z
      Skip s'   → loop_length z s'
      Yield _ s' → loop_length (z + 1) s'
```